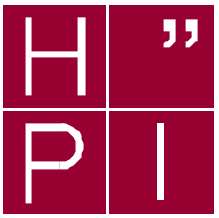
HASSO-PLATTNER-INSTITUT
für Softwaresystemtechnik an der Universität Potsdam



Visualizing Design and Spatial Assembly of Interactive CSG

Florian Kirsch
Marc Nienhaus
Jürgen Döllner

Technische Berichte Nr. 7
des Hasso-Plattner-Instituts
für Softwaresystemtechnik an der Universität Potsdam

Technische Berichte des Hasso-Plattner-Instituts für Softwaresystemtechnik
an der Universität Potsdam

Nr. 7

Visualizing Design and Spatial Assembly of Interactive CSG

Florian Kirsch
Marc Nienhaus
Jürgen Döllner

Potsdam 2005

Bibliografische Information der Deutschen Bibliothek
Die Deutsche Bibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.ddb.de abrufbar.

Die Reihe *Technische Berichte des Hasso-Plattner-Instituts für Softwaresystemtechnik an der Universität Potsdam* erscheint aperiodisch.

Herausgeber: Professoren des Hasso-Plattner-Instituts für Softwaresystemtechnik an der Universität Potsdam

Redaktion: Marc Nienhaus, Jürgen Döllner
Email: {marc.nienhaus, juergen.doellner }@hpi.uni-potsdam.de

Vertrieb: Universitätsverlag Potsdam
Postfach 60 15 53
14415 Potsdam
Fon +49 (0) 331 977 4517
Fax +49 (0) 331 977 4625
e-mail: ubpub@rz.uni-potsdam.de
http://info.ub.uni-potsdam.de/verlag.htm

Druck: allprintmedia gmbH
Blomberger Weg 6a
13437 Berlin
email: info@allprint-media.de

Heft 7 (2005)
ISBN 3-937786-56-2
ISSN 1613-5652

Visualizing Design and Spatial Assembly of Interactive CSG

Florian Kirsch Marc Nienhaus Jürgen Döllner

Abstract

For interactive construction of CSG models understanding the layout of a model is essential for its efficient manipulation. To understand position and orientation of aggregated components of a CSG model, we need to realize its visible and occluded parts as a whole. Hence, transparency and enhanced outlines are key techniques to assist comprehension.

We present a novel real-time rendering technique for visualizing design and spatial assembly of CSG models. As enabling technology we combine an image-space CSG rendering algorithm with blueprint rendering. Blueprint rendering applies depth peeling for extracting layers of ordered depth from polygonal models and then composes them in sorted order facilitating a clear insight of the models. We develop a solution for implementing depth peeling for CSG models considering their depth complexity. Capturing surface colors of each layer and later combining the results allows for generating order-independent transparency as one major rendering technique for CSG models. We further define visually important edges for CSG models and integrate an image-space edge-enhancement technique for detecting them in each layer. In this way, we extract visually important edges that are directly and not directly visible to outline a model's layout. Combining edges with transparency rendering, finally, generates edge-enhanced depictions of image-based CSG models and allows us to realize their complex, spatial assembly.

Categories and Subject Descriptors: I.3.3 [Computer Graphics]: Picture/Image Generation – Display Algorithms, I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling – Constructive solid geometry

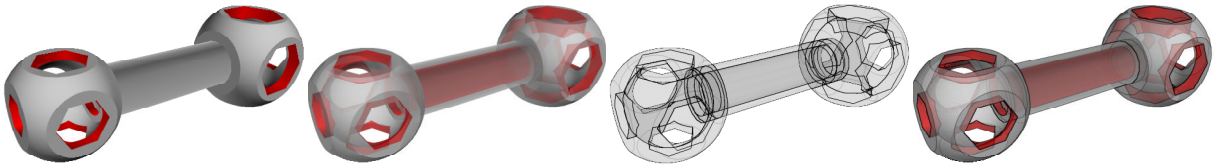


Figure 1: *A CSG model of a spanner rendered with an image-space CSG rendering technique (a), our transparency rendering technique (b), our blueprint rendering technique (c), and our edge-enhanced transparency technique (d).*

1 Introduction

Constructive Solid Geometry (CSG) represents a fundamental concept for modeling 3D solids, e.g., complex aggregate objects such as mechanical components. In interactive applications for constructing CSG models the user requires both: a constant visual feedback to facilitate interactive composition of various transient CSG components as well as a comprehensible visualization of the CSG models' design and spatial assembly.

Image-based CSG rendering algorithms synthesize a graphical representation of CSG models in real-time without calculating their triangulation. In contrast to object-based CSG algorithms, which involve high computational costs due to the required triangulation, image-based CSG is best suited for user interfaces supporting direct manipulation of CSG models.

Depictions produced by image-based CSG rendering algorithms generally illustrate CSG models by shaded geometry represented only by the outer surface of the solids (Figure 1a). As a result, inner parts of the CSG models are not visible and, therefore, the entire assembly becomes increasingly difficult to understand during the modeling process. In particular, complex assemblies occur in most CAD/CAM models.

Non-photorealistic rendering (NPR) focuses on generating vivid and expressive depictions that facilitate visual perception. For instance, Diepstraten et al. [DWE02] introduce a concept for view-dependent transparency in the field of NPR to generate technical illustrations of polygonal models. As common technique in NPR, enhanced outlines of single features assist comprehension.

Inspired by classical technical illustrations, we have developed a real-time rendering technique that generates transparent, edge-enhanced depictions of CSG models. It provides a solution for depth peeling of image-based CSG

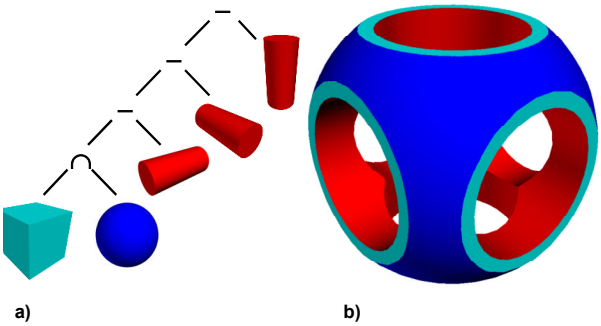


Figure 2: *The CSG tree (a) shows a partial-product of a sphere and a box from which cylinders are abstracted. This results in the CSG model of the widget (b).*

that extracts layers of ordered depth. The layers are then composed to implement order-independent transparency rendering of CSG models (Figure 1b). Although these depictions provide an insight into CSG models, outlines of single features are still hardly noticeable, in particular in regions of high depth complexity. Therefore, we extract visually important edges of outer and inner parts of a CSG model in image space using blueprint rendering. The resulting blueprint-like depictions represent expressive and meaningful illustrations, which outline inner and outer features (Figure 1c). Finally, we combine both techniques to enrich order-independent transparency rendering by enhanced outlines (Figure 1d). The resulting images illustrate layout and relationships of CSG components very precisely and, hence, communicate the spatial assembly of CSG models in a very clear and convincing way.

The remainder of this paper is structured as follows: Section 2 discusses related work, Section 3 describes depth peeling in general, and Section 4 introduces depth peeling for CSG models to implement order-independent transparency. Section 5 presents edge-enhanced transparent depictions of CSG models. Section 6 provides some details about our implementation and the resulting performance, and Section 7 discusses future work.

2 Related Work

CSG modeling means defining 3D geometry as result of set operations ($\cup$, $\cap$, $-$), applied to basic, closed 3D primitives or to other CSG geometry defined in this way. For specifying a CSG shape the CSG tree represents the fundamental structure (Figure 2a and 2b).

Generating a 3D polygonal model of a CSG shape's surface is computationally expensive and for complex models not possible in real-time. Hence, in interactive applications for modeling CSG geometry, rendering CSG models using an object-space approach is not appropriate.

Goldfeather et al. developed the first algorithm for image-based rendering of CSG [GMT*89], that is, rendering CSG images without calculating the final 3D geometric mesh. An important part of their work is the normalization of arbitrary CSG trees into an equivalent union-of-partial-product form. Visibility of partial products (also called CSG products) can be effectively determined by image-based graphics hardware operations, and still today all image-based CSG algorithms require the normalization step.

Wiegand described the first adaptation of the algorithm of Goldfeather for OpenGL [Wie96]. He used an emulation of two z-buffers, one for calculating the visibility of a single primitive and another to combine the results. This approach was improved by real-time capable, texture-based approaches, independently developed by Kirsch and Döllner transferring visibility information in color textures [KD04], and Guha et al., who apply depth peeling and thus do not require a second depth buffer [GKM*03]. Note that Guha et al. did not use depth peeling for determining inner depth layers of CSG shapes, as we describe in this paper.

Non-photorealistic rendering (NPR) has become a core discipline in computer graphics over the last decades and increasingly contributes to the field of visualization. Visualization strategies using NPR exist, for instance, for volume rendering and medical visualizations [LME*02, RE01, TC00], and for technical and architectural visualizations [GSG*99, DWE02, ND04]. In this spirit, we introduce a novel NPR technique that is capable of visualizing the design and the spatial assembly of CSG models efficiently by enhancing visible and occluded visually important edges.

In general, object-space and hybrid silhouette rendering algorithms allow for rendering stylized edges and they can be applied to occluded edges as well [KDM*03]. The problem is that they operate on the given polygonal mesh of a 3D model, which cannot be provided by interactive image-space CSG rendering algorithms. In contrast, image-space edge enhancement algorithms are generally independent from the polygonal mesh but cannot be extended to occluded edges directly. Isenberg et al. [IFH*03] provide a comprehensible survey on silhouette extraction algorithms.

For enhancing visible and occluded edges of CSG models, we use the blueprint rendering technique [ND04]. It applies an image-space edge-enhancement algorithm [ND03] that extracts visually important edges in real-time by detecting discontinuities in G-buffers, the normal and the z-buffer [ST90]. Blueprint rendering further deploys depth peeling, an image-space technique for extracting layers of ordered depth from a 3D model. Mammen [Mam89] initially implemented high-quality antialised transparency rendering of 3D models by processing pixels in depth-sorted order using the Virtual Pixel Maps architecture. Diefenbach [Die96] extended this approach to general hardware by introducing the dual z-buffer concept for rendering fragments in depth-sorted order. Finally, Everitt [Eve01] implemented the dual z-buffer on common graphics hardware to facilitate depth peeling and order-independent transparency rendering of 3D models in real-time.

3 Peeling Away Layers of Unique Depth Complexity

Depth peeling is a multipass rendering technique that operates on a per-fragment basis and extracts 2D layers of 3D

geometry in depth-sorted order. Generally speaking, depth peeling successively "peels away" layers of unique depth complexity.

In general, fragments passing an ordinary depth test define the minimal z-value at each pixel location. But we cannot determine the fragment that comes second (or third, etc.). Hence, we need an additional depth test to extract those fragments that form a layer of a given ordinal number with respect to depth complexity. In this way, we extract the first *n* layers by *n* rendering passes.

We denote a layer of unique depth complexity as a *depth layer* and a high-precision texture received from capturing the associated z-buffer contents as a *depth layer map*. Accordingly, we call the contents of the associated color buffer captured in an additional texture a *color layer map*. In particular, color layer maps can later be used in depth-sorted order to compose the final rendition. Figure 3 shows color layer maps and depth layer maps of a 3D model of consecutive depth layers.

For depth peeling of polygonal models, we follow the implementation given by Nienhaus and Döllner [ND04]. We render a 3D model multiple times performing two depth tests on each fragment produced by the rasterizer. If no fragment gets rendered into the frame buffer, depth peeling terminates; otherwise it continues with the next depth layer. That is, if the number of rendering passes has reached the maximum depth complexity of the model, all its depth layers have been peeled away.

3.1 Performing Two Depth Tests

In the first rendering pass, we perform an ordinary depth test on each fragment. We capture the contents of the z-buffer and the color buffer in a depth layer map and a color layer map.

In consecutive rendering passes, we perform an additional depth test on each fragment. For this test, we use the depth layer map of the previous rendering pass. Thereby, we project the texture onto the 3D model by generating texture coordinates in such a way that they correspond to canvas coordinates of the targeted pixel position. In this way, a texture access can provide a fragment with the z-value stored at that pixel position in the z-buffer of the previous rendering pass.

The two depth tests work as follows:

- If the current z-value of a fragment is greater than the corresponding texture value of the depth layer map, the fragment proceeds and the second, ordinary depth test is performed.
- Otherwise, if the test fails, the fragment gets rejected.

When we have processed all fragments, the contents of the z-buffer and the color buffer form the new depth layer map and color layer map. We can efficiently implement the additional depth test using fragment shaders. Furthermore, we use occlusion queries [Kil04] to implement the termination condition.

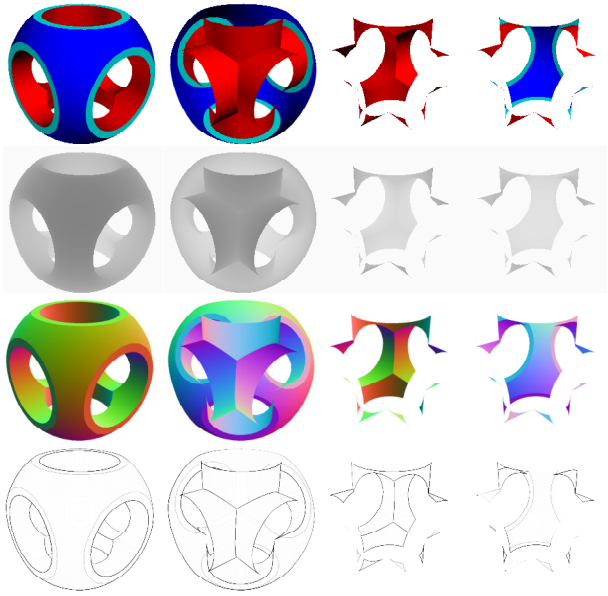


Figure 3: *The color layer maps (first row) of each depth layer (column) can be used for transparency rendering of CSG shapes. Discontinuities in the depth layer map (second row) and the normal buffer (third row) form the resulting edge maps (forth row) for each layer.*

4 Depth Peeling for CSG Rendering

Depth peeling does not directly map to image-based CSG rendering, which substantially uses the fact that CSG primitives are only partly visible and calculates the visible areas in image-space. Therefore, depth layers of the final CSG shape cannot just be determined by linearly traversing all CSG primitives for each layer. Instead, CSG calculations are required for each depth layer.

4.1 Terms and Definitions

For depth peeling of CSG models respectively of a CSG product, we denote surfaces of the model that face towards the viewer as *front surface*, and that face backwards from the viewer as *back surface*. Both front and back surfaces form the *surface* of a CSG model. For instance, for non-transparent CSG rendering, only the nearest front surface is visible. Inner depth layers of a CSG model can consist both of front and back surfaces.

Image-based CSG rendering relies on determining parts of CSG primitives that are *visible*, that is, are part of the surface of the CSG product they are contained in. An intermediate step are *potentially visible* parts of a CSG primitive, which are a superset of visible parts and can be determined with a simple cull-face operation: Within the front surface of a CSG product, front-facing polygons of intersected and back-facing polygons of subtracted primitives are potentially visible. In the back surface, back-facing polygons of intersected and front-facing polygons of sub-

```
function cullMode(P ← CSG primitive, mode ← {front, back})
begin
    if (P.isIntersected and mode=front) return cullBackFaces
    if (P.isSubtracted and mode=front) return cullFrontFaces
    if (P.isIntersected and mode=back) return cullFrontFaces
    if (P.isSubtracted and mode=back) return cullBackFaces
end

procedure calcCSG (G ← CSG product,mode ← {front, back})
begin
    for (P ← each CSG primitive in G)
        enable temporary frame buffer
        F ← rasterize(P) with cullMode(P, mode)
        /* potentially visible parts of P */
        for (f ← each fragment in F)
            if (i=1 or f.depth >value_depth layer map(i-1))
                f.depth → z-buffer
                visible → visibility channel
        end for
        /* determine visible parts of P with parities */
        for (Q ← each CSG primitive in G \ P)
            // calculate parity for all fragments of Q
            // if fragment f in F is not visible due to parity
                not visible → visibility channel
        end for
        /* visibility transfer */
        visibility map ← capture(visibility channel)
        enable main frame buffer
        for (f ← each fragment in F)
            if (i=1 or f.depth >value_depth layer map(i-1))
                if (value_visibility map is visible)
                    f.depth → z-buffer
        end for
end

procedure CSGDepthPeeling(S ← CSG shape)
begin
    i ← 1
    do
        for (G ← each CSG product in S)
            calcCSG(G, front)
            if (i>1)
                calcCSGt(G, back)
        end for
        depth layer map(i) ← capture(z-buffer)
        i ← i+1
    while (fragments were rendered in main frame buffer)
end
```

Listing 1: *Pseudo code illustrating our implementation of depth peeling for CSG shapes.*

tracted primitives are potentially visible. The complicated part of image-based CSG rendering is to sort out potentially visible parts of a CSG primitive that are not visible.

A *visibility channel* is a color channel that holds binary information whether a CSG primitive is visible or not. As described in [KD04], texture maps holding visibility channels can be used efficiently for the *visibility transfer*, i.e., to assemble visible parts of different CSG primitives in the frame buffer: We project the *visibility map* onto visible parts of the CSG primitive using a "z-less" test. Thereby, with the alpha-test only fragments that pass are marked as visible in the visibility map.

4.2 Determining the front CSG depth layer

Until now all algorithms for image-based CSG have only determined the front most visible depth layer of a CSG shape. In this section, we describe the basic CSG algorithm, which we later enhance to support depth peeling. We restrict our description to convex primitives because the extensions to support concave primitives for CSG depth peeling do not differ from normal CSG algorithms at all.

For each CSG primitive P in a partial product, we calculate separately which parts are within the front surface of the product and, hence, are visible. For this, in a temporary off-screen buffer we render the potentially visible part of P into the depth buffer and we also mark this area of P in a visibility channel. Then, for each other primitive Q in the partial product we calculate the *parity*, i.e., the number of depth layers of Q in front of P, using stencil-inversion operations. For visible parts of P, the parity must be odd for intersected Q and even for subtracted Q. Hence, the parity is used to filter out parts of P, which are invisible due to Q. Those areas are marked as invisible in the visibility channel. After the parity has been calculated for all Q, the visibility channel encodes visible parts of P. The channel is used as texture for the visibility transfer, i.e., to regenerate the depth values of visible parts of P in the main depth buffer.

After all primitives in all partial products have been processed this way, the main depth buffer contains the first depth layer of the CSG shape. The respective depth layer map is used for depth peeling in the next rendering pass.

4.3 Determining inner CSG depth layers

Inner depth layers of a CSG shape can consist of both front and back surfaces of CSG products. For instance, consider the second depth layer: It can consist of the back surface of the partial product of which the front surface formed the first depth layer. But it can also consist of the front surface of a second partial product that penetrates the first CSG product.

Therefore, for inner depth layers we do the CSG calculation twice for each primitive, once to determine visible parts in the front surface of the CSG product and once for the back surface. A CSG calculation for a primitive works similarly as described in Section 4.2, but with some additions: Potentially visible areas are determined with respect to the depth layer map of the former depth layer, i.e., only fragments behind the depth layer map are considered to be potentially visible. In the same way, during the visibility transfer only fragments behind the depth layer map are generated.

The parity test, on the other hand, is done without depth peeling. It does also not depend whether we test visibility in the front or in the back surface: Because no surface of a primitive in a CSG product can be located between the front and next back surface of the product, the result of the parity does not differ for these neighboring surfaces. Consequently the parity test can be always applied in the same way.

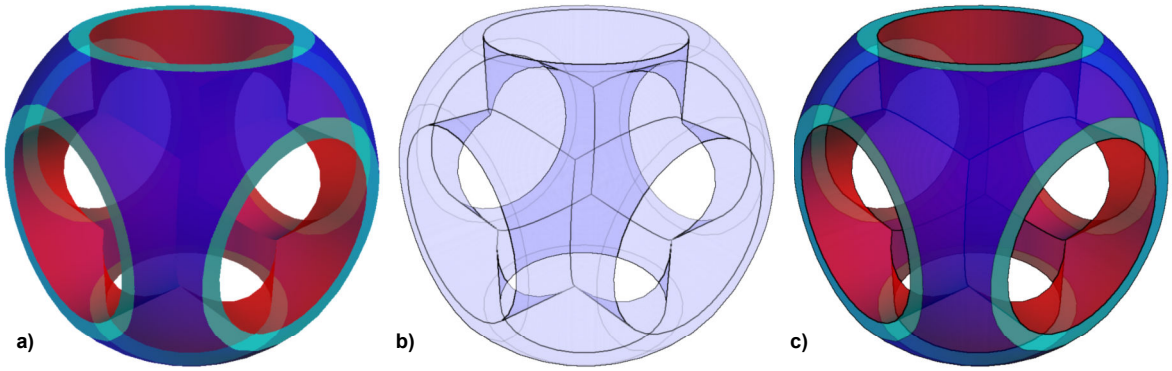


Figure 4: *A transparency rendering of a CSG shape (a). A blueprint as an edge enhanced depth complexity cueing visualizes the design of a CSG shape (b). An edge-enhanced transparency rendering of a CSG shape visualizes the design as well as its spatial assembly (c).*

Listing 1 outlines the entire depth-peeling algorithm for image-based CSG.

4.4 Optimization for single CSG products

The algorithm described above requires two CSG calculations for each primitive and for all depth layers except the first. Since this is rather expensive, it is worthwhile to optimize the common case of only a single CSG product (Listing 2). In this case, we can profit from the fact that, for consecutive depth layers, front surfaces of the CSG product alternate with back surfaces. This means that potentially visible polygons in the second, and in all even depth layers are exactly those polygons that are not potentially visible in the first depth layer. More specifically, in even depth layers only the back faces of intersected primitives and the front faces of subtracted primitives are potentially visible.

```
procedure backCSG(G ← CSG product)
begin
    for (P ← each CSG primitive in G)
        F ← rasterize(P) with cullMode(P, back)
        for (f ← each fragment in F)
            if (f.depth < z-buffer
                and f.depth > value_depth layer map(i-1))
                f.depth → z-buffer
        end for
    end for
end

procedure CSGDepthPeeling(G ← CSG product)
begin
    i ← 1
    do
        if (i is odd)
            calcCSG(G, front)
        else        /* i is even */
            backCSG(G)
        depth layer map(i) ← capture(z-buffer)
        i ← i+1
    while (fragments were rendered in main frame buffer)
end
```

Listing 2: *Depth peeling for a single CSG product.*

Therefore, to determine an even depth layer, we only need to find the nearest back-facing polygons of intersected and front-facing polygons of subtracted primitives behind the formerly calculated depth layer map. Since we already know that the former depth layer is a front surface of the CSG product, it is clear that these polygons limit the extent of the CSG product backwards.

To determine the third and greater odd depth layers, we do a CSG calculation and apply the parity test for all primitives again. But since odd depth layers can only consist of the front surface, only front facing polygons of intersected respectively back facing polygons of subtracted primitives are potentially visible, as in the case of the first depth layer. Additionally, only fragments behind the former, equal depth layer map are considered to be potentially visible. The parity then is calculated and used for determining visibility as before.

4.5 Transparency Rendering of CSG Models

In addition to writing depth values of visible parts of primitives we can also write shaded colors into the color buffer during the visibility transfer. The resulting depth layer and color layer maps for each depth layer suffice to generate a depth correct depiction of a transparent CSG shape. For this, we render the color layer maps as depth sprites in depth-sorted order from back to front. Thereby, for each depth layer, we texture a screen-aligned quad covering the whole viewport of the canvas with the according textures as input. A specialized fragment shader then replaces the depth value of each fragment of the quad by the correspondent texture value of the depth layer map. Additionally, the shader replaces the color value and the alpha value of each fragment by the correspondent texture value of the color layer map. In this way, we blend each depth sprite with the contents of the frame buffer using alpha blending. Figures 1b and 4a illustrate the resulting rendering of transparent CSG models.

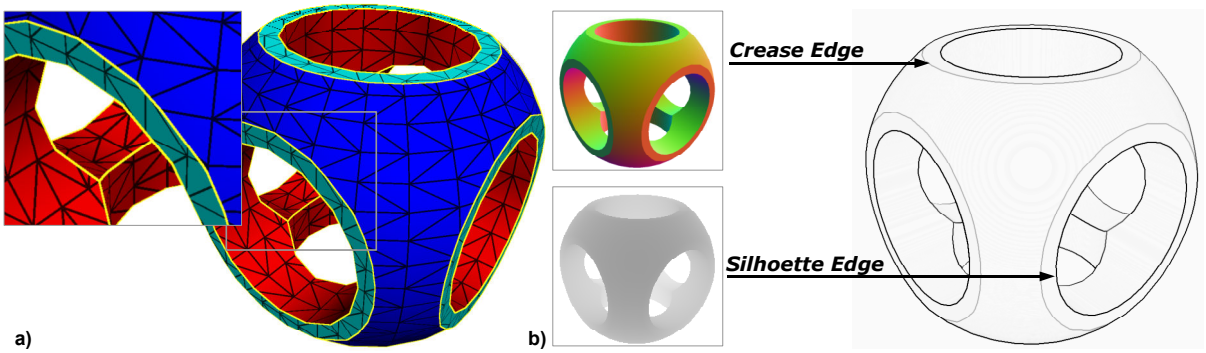


Figure 5: *Generally, visually important edges of a CSG model do not correspond to edges of the polygonal mesh (a). Crease and silhouette edges represent discontinuities in the normal and z-buffers and are assembled as edge intensities in the edge map (b).*

5 Edge-Enhanced Depictions of CSG Models

For visualizing the design of CSG models in a more explicit way, we use blueprint rendering to enhance their visually important edges that are directly visible or that become visible when peeling away depth layers.

5.1 Visually Important Edges

We consider silhouette edges and crease edges [ND03] of a closed 3D model as visually important and enhance these edges to increase the perception of the model.

With respect to polygonal 3D geometries, each visually important edge corresponds to edges of the mesh that connect vertices: A silhouette edge is an edge adjacent to one front facing polygon and one back facing polygon; a crease edge is adjacent to two front facing polygons whose dihedral angle is below some threshold.

Set operations applied to CSG primitives for composing CSG models produce surfaces whose edges cannot be represented by edges of the original polygonal meshes in general. Furthermore, image-space CSG algorithms clip part of the mesh without adding new polygons. In conclusion, visually important edges of the resulting CSG model do not necessarily correspond to edges of the mesh of one of its CSG primitives. Figure 5a exemplifies that visually important edges of the final CSG model (yellow) are obviously independent from edges of its underlying meshes (black).

Therefore, we need to alter our definition of visually important edges. Hence, *silhouette edges* of CSG shapes represent edges where part of a front surface joins part of a back surface. *Crease edges* represent edges where two front surfaces or two back surfaces of CSG primitives join and form a certain angle. Figure 5b illustrates silhouette and the crease edges that are produced by set operations.

An image-space edge enhancement algorithm is capable of extracting visually important edges of 3D models regardless of the constitution of their meshes. The algorithm obtains silhouette edges by detecting discontinuities in the z-buffer and crease edges by detecting discontinuities in the normal buffer. For this, z-values and encoded per-fragment normal values are rendered directly into textures. Then, we texture a screen-aligned quad using these textures. Sampling neighboring texels allows us to extract discontinuities that result in intensity values. Finally, the assembly of intensity values constitutes edges that we render directly into a single texture, called an *edge map*. Figure 5b depicts the normal and z-buffer and the resulting edges.

5.2 Visible and Not Visible Edges

We denote visually important edges that are directly seen by the virtual camera as *visible edges*. In contrast, *not visible edges* represent visually important edges that are occluded by faces of the 3D model, i.e., they are not directly seen.

We complement our original depth-peeling algorithm for CSG models by the edge-enhancement algorithm to extract visually important edges for each depth layer. Since discontinuities in the normal buffer and z-buffer constitute visible edges we have to construct both in each rendering pass. We encode per-fragment normal values as color values to generate the normal buffer as color layer map. Then, we can directly construct the edge map, because the depth layer map already represents the depth buffer. In conclusion, not visible edges become visible when peeling away depth layers successively and can then be extracted in the same manner.

As a result, our technique preserves visible and not visible edges as edge maps in depth-sorted order for further enhancements. Figure 3 shows z-buffers, normal buffers, and resulting edge maps of the CSG model produced for successive depth layers.

5.3 Blueprint Composition

When edge maps have been produced for all depth layers, we start composing blueprints by rendering visible and not visible edges in back-to-front order. In contrast to transparency rendering, we now render edge maps of each depth layer as depth sprites. For this, we use the according depth layer map and edge map as input. Besides replacing depth values, the modified fragment shader replaces the color value of each fragment by the intensity value of the edge map. Furthermore, rejecting fragments that do not contrib-

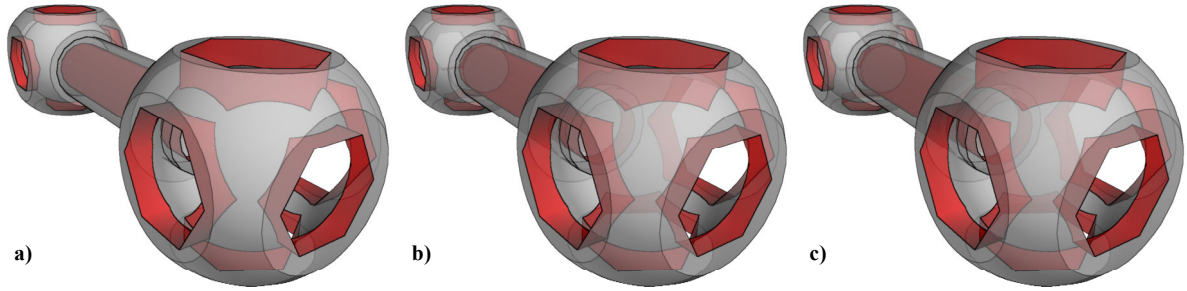


Figure 6: *Edge enhanced transparency renderings of the spanner for which two depth layers (a), three depth layers (b), and four depth layers (c) have been considered. The visual quality of b) and c) in showing details is nearly identical.*

ute to edges (determined via a threshold value) generates a wire-frame depiction of the CSG model produced by visible and not visible visually important edges only.

For depth complexity cueing while keeping edges enhanced, we, alternatively, blend each depth sprite with the frame buffer contents. For this, the shader replaces the alpha value of each fragment by the edge intensity and replaces its color value by the product of the intensity value and a constant, e.g., a bluish color.

Either way, blueprints can easily be merged with further 3D scene contents because depth values of the original CSG model are used.

Figures 1c and 4b depict the final blueprints of CSG models using edge-enhanced depth complexity cueing.

5.4 Edge-Enhanced Transparency Rendering

Composing visible and not visible edges of a CSG model outlines its design. But its spatial assembly based on several CSG primitives is still difficult to perceive because visual indicators such as surface shading, which facilitate identifying single components and their orientation in 3D space, are missing. In contrast, transparency renderings provide surface shading, but their outlines are hardly noticeable, in particular in regions of high depth complexity. Therefore, we enhance transparency renderings of CSG models by additionally accentuating visible and not visible edges.

For this, we have to synthesize two different color layer maps for each depth layer: One color layer map for preserving the surface shading of the CSG model and one color layer map for preserving the normal buffer of the CSG model. We create both textures simultaneously using multiple render targets [Kil04], i.e., we write into several color buffers at once, and reuse the results as texture maps.

As before we can construct the edge map for each depth layer. Then, we combine the edge map with the color layer map that contains the surface shading when rendering each depth layer as depth sprite. In particular, we multiply intensity values of the edge maps by color values of the color layer map. We then blend each layer with the frame buffer contents using alpha values of the color layer map as blend factors. Figures 1d and 4c illustrate resulting edge-enhanced transparency renderings of CSG models.

6 Implementation and Performance

Our rendering technique runs at interactive frame rates on today's graphics cards. It fully exploits its capabilities: It is based on the OpenGL Shading Language [Ros04], uses multiple render targets, and dynamically generates multiple intermediate rendering results as textures for storing visibility maps, depth layer maps, color layer maps, and edge maps.

A fundamental graphics operation is depth peeling to process all depth layers of a CSG model up to its depth complexity. In practice, the first few layers are typically sufficient. The remaining depth layers have less visually impact to the overall composition because only few, often isolated pixels are produced. In order to increase performance, we restrict the number of rendering passes. Depth peeling terminates if fewer than a specified (desired) minimal number of fragments are rendered for the CSG model. This way, we decrease the number of rendering passes while we maintain visual quality of our depictions. The tradeoff between speed and quality can easily be implemented by configuring the occlusion query extension. Figure 6 depicts the spanner using different numbers of depth layers.

Besides the polygon count, the performance of our rendering technique is essentially bound to (1) the depth complexity respectively the number of rendering passes, (2) the window resolution, (3) the number of CSG primitives, and (4) the layout of the normalized CSG tree, respectively, the set operations that are used.

The widget in Figure 4 takes 38.6 fps for the transparency rendering, 30.4 fps for the blueprint depiction, and 30.1 fps for the edge-enhanced transparency rendering while considering 4 depth layers at a window resolution of 512×512 on a GeForce 6800 GT. Since the widget only contains a single CSG product these numbers are for the optimized algorithm for CSG depth peeling of a single CSG product. The spanner in Figure 1 takes 9.2 fps for the transparency rendering, 8.6 fps for the blueprint depiction, and 8.1 fps for the edge enhanced transparency rendering in the same environment. It is notable that the spanner takes 8.7, 7.1, and 7.0 fps for rendering at a window resolution of 1024×1024, i.e., the rendering performance decreases only slightly with higher resolution.

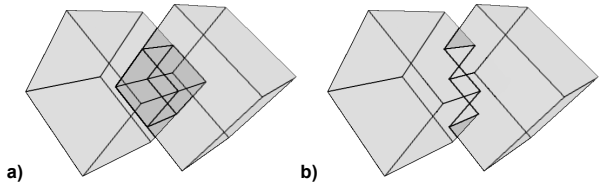


Figure 7: *Two different visual interpretations of a union of CSG products: Each part rendered independently (a); both part rendered as a single volume (b).*

7 Conclusions

We have presented a novel real-time rendering technique for visualizing, illustrating, and outlining in a perceivable way design and spatial assemblies of image-based CSG models. It is implemented on top of OpenGL, and its results can be merged with arbitrary 3D scene contents. Thus, one can integrate this technique into any real-time modeling and rendering framework, for example, to provide visual feedback and insight when constructing CSG models.

Our approach handles unions of CSG products in a way that is not the only possible one: Depth layers of different CSG products that intersect are calculated and visualized independently. Therefore, inside the volume of a CSG product, visually important edges of other products are visible (Figure 7a). Another approach would be to visualize only the intersecting edges of the borders of different CSG products and not to follow the extent of participating CSG products inside other CSG products (Figure 7b). This method may feel more suitable for CSG, which guarantees that CSG shapes are solid and form a volume. But for blueprint rendering our method is able to emphasize spatial relations of different CSG products even more, as it displays the common area of different CSG products in a different tone. Nonetheless, more research is required in this direction.

References

[Die96] DIEFENBACH P. J.: Pipeline Rendering: Interaction and Realism Through Hardware-based Multi-Pass Rendering. *Ph.D. thesis*. University of Pennsylvania, June 1996.

[DWE02] DIEPSTRATEN J., WEISKOPF D., AND ERTL T.: Transparency in Interactive Technical Illustrations. *Computer Graphics Forum 21*, 2 (Sept. 2002), C317-C325. (Proc. Eurographics'02).

[Eve01] EVERITT C.: Interactive Order-Independent Transparency. *Technical report*. NVIDIA Corporation, 2001.

[GKM*03] GUHA S., KRISHNAN S., MUNAGALA K., AND VENKATASUBRAMANIAN S: Application of the Two-Sided Depth Test to CSG Rendering. In *Proc. of the 2003 Symposium on Interactive 3D Graphics*, 177-180.

[GMT*89] GOLDFEATHER J., MOLNAR S., TURK G., AND FUCHS H.: Near Realtime CSG Rendering Using Tree Normalization and Geometric Pruning. In *IEEE Computer Graphics and Applications 9*, 3 (May 1989), 20-28.

[GSG*99] GOOCH B., SLOAN P.-P. J., GOOCH A., SHIRLEY P., RIESENFELD R.: Interactive Technical Illustration. In *Proc. of the 1999 Symposium on Interactive 3D Graphics*, 31-38.

[IFH*03] ISENBERG T., FREUDENBERG B., HALPER N., SCHLECHTWEG S., AND STROTTHOTTE T.: A Developer's Guide to Silhouette Algorithms for Polygonal Models. In *IEEE Computer Graphics and Applications 23*, 4 (July/August 2003), 28-37.

[KD04] KIRSCH F. AND DÖLLNER J.: Rendering Techniques for Hardware-Accelerated Image-Based CSG. In *Journal of WSCG'04*, 221-228.

[KDM*03] KALNINS R. D., DAVIDSON P. L., MARKOSIAN L, AND FINKELSTEIN A.: Coherent Stylized Silhouettes. In *ACM TOG 22*, 3 (July 2003), 856-861. (Proc. of ACM SIGGRAPH 2003).

[Kil04] KILGARD M. (ED.): NVIDIA OpenGL Extension Specifications. *NVIDIA Corporation*, December 2004.

[LME*02] LU A., MORRIS C., EBERT D., RHEINGANS P., AND HANSEN C.: Non-photorealistic Volume Rendering Using Stippling Techniques, In *Proc. of IEEE Visualizaton 2002*, 211-218.

[Mam89] MAMMEN A.: Transparency and Antialiasing Algorithms Implemented with the Virtual Pixel Maps Technique. In *IEEE Computer Graphics and Applications 9*, 4 (July 1989), 43-55.

[ND03] NIENHAUS M. AND DÖLLNER J.: Edge Enhancement – An Algorithm for Real-Time Non-Photorealistic Rendering. In *Journal of WSCG'03*, 346-353.

[ND04] NIENHAUS M. AND DÖLLNER J.: Blueprints – Illustrating Architecture and Technical Parts using Hardware-Accelerated Non-Photorealistic Rendering. In *Proc. of Graphics Interface 2004*, 49-56.

[RE01] RHEINGANS P. AND EBERT D.: Volume Illustration: Non-Photorealistic Rendering of Volume Models. In *IEEE Transactions on Visualization and Computer Graphics 7*, 3 (2001), 253-264.

[Ros04] ROST R. T.: *OpenGL Shading Language*. Addison-Wesley, 2004.

[ST90] SAITO T. AND TAKAHASHI T.: Comprehensible Rendering of 3-D Shapes. In *Proc. of ACM SIGGRAPH 1990*, 197-206.

[TC00] TREAVETT S. M. F. AND CHEN M.: Pen-and-Ink Rendering in Volume Visualisation. In *Proc. IEEE Visualization 2000*, 203-210.

[Wie96] WIEGAND T. F.: Interactive Rendering of CSG Models. In *Computer Graphics Forum 15*, 4 (1996), 249-261.

ISBN 3-937786-56-2
ISSN 1613-5652